Hybrid systems: a real-time interface to control engineering

Eriksen, Thomas Juul; Heilmann, Søren; Holdgaard, Michael; Ravn, Anders P.

Published in:
Proceedings of 8th Euromicro Workshop on Real-Time Systems

Link to article, DOI:
10.1109/EMWRTS.1996.557830

Publication date:
1996

Document Version
Publisher's PDF, also known as Version of record

Hybrid Systems: A Real-Time Interface to Control Engineering

Thomas J. Eriksen, Søren T. Heilmann, Michael Holdgaard & Anders P. Ravn
Department of Information Technology
Technical University of Denmark
DTU Building 344, DK-2800 Lyngby, Denmark
apr@it.dtu.dk
*

Abstract

This paper gives an introduction to a general hybrid systems model for definition of system requirements and a corresponding software architecture together with an example of their specialization for use in implementing a mode-switching controller for a hydraulic cylinder.

1. Introduction

An important application area for real-time computing is embedded systems where the computing system provides intelligent control of a mechanical, chemical etc. plant or device. The software requirements for such applications depend heavily on the properties of the plant. These properties are usually investigated by control engineers that base their work on the theory of dynamic systems [3, 9]. The mathematical tool for this work is thus mathematical analysis, in particular the theory of differential equations.

If the models are linear differential equations, there are standard techniques for implementing a corresponding discrete, sampled system in the form of a computer program, and the work for the software developer is reduced to ensuring proper timing of periodic processes in a selected architecture using appropriate scheduling theory, see e.g. [2].

However, when the models include non-linear systems, there are only in special cases ready recipes for the synthesis of control algorithms. A promising approach is here to see the plant and the computer as a hybrid system where a piecewise continuous dynamic system interacts aperiodically with a state machine or automaton implemented by the computer program. This approach is the focus of much research by control theorists and computer scientists in recent years [7, 12].

*Research partially funded by the Danish Research Councils.

Design and implementation of a hybrid controller means that control engineers and programmers need to cooperate. The control engineers must find an appropriate hybrid model for the system, see e.g. [4]. This includes an identification of the plant characteristics and selection of regulators. The programmers must implement the controller that switches between the regulators when the continuous component of the model satisfies certain conditions. In essence the program is a phase transition system with important events [10].

In order to investigate the facets of such a cooperation we have engaged in a series of experiments with a concrete system that uses mode switched control to improve performance [8, 1].

In this paper we report on our motivation for and implementation of a generic software framework for distributed mode-switching control programs. The proper design and construction of such a framework is the contribution of the software engineer. The choice of generic parameters for the framework is a concrete manifestation of the interface between software and control engineering. The contribution of the control engineer is to provide actual parameters for instantiation of the framework to form a desired hybrid controller of a given plant.

The paper gives an introduction to a hybrid systems model in Section 2, a corresponding software architecture in Section 3, and discusses implementation issues in the concluding Section 4.

2. Hybrid systems

In this section we will describe a model for the hybrid systems supported by our software framework and subsequently introduce and motivate the choice of parameters of the framework.

1068-3070/96 $5.00 © 1996 IEEE
Proceedings of EURWRTS '96

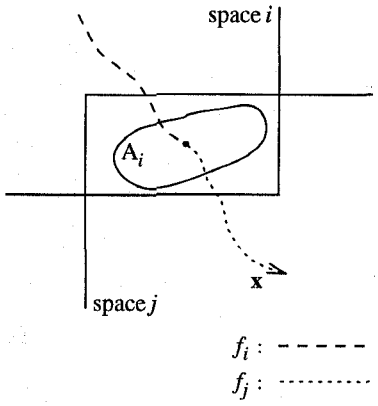

Figure 1. An autonomous switch of model

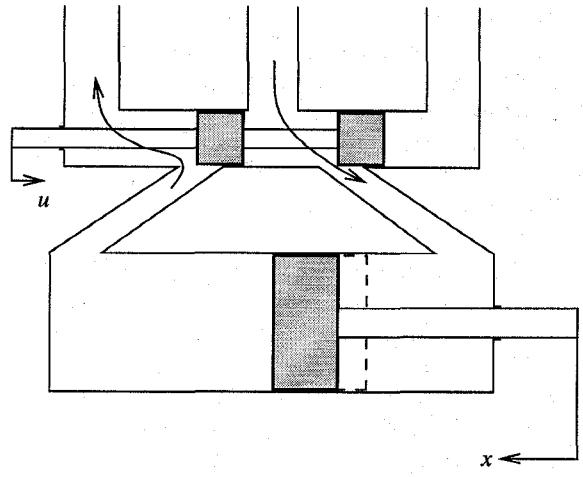

Figure 2. A hydraulic actuator

2.1. Model

When approaching a control problem the control engineer will usually start of by constructing a mathematical model of the system, the *plant*, to be controlled. Taking the state based view, this model will often take the form of a set of first order differential equations derived from the physics of the plant and governing the evolution of the state:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)) \quad (1)$$

where $\mathbf{x}(t)$ is the *continuous component* of the state with values in a subset of an Euclidean space. The *controlled vector field* f generally depends on $\mathbf{x}(t)$ and the *continuous component* $\mathbf{u}(t)$ is the control input.

A hybrid system, cf. Branicky, Borkar and Mitter [4], is modeled by an indexed, countable family of ordinary models:

$$\dot{\mathbf{x}}(t) = f_i(\mathbf{x}(t), \mathbf{u}(t)) \quad , i \in \mathbb{N} \quad (2)$$

where the index i determines a new model possibly even in a state space with changed dimensions.

When a transition from one model to another occurs, the trajectory may either jump or stay continuous (switch, in the terminology of [4]). A transition may either be autonomous, caused by the trajectory hitting a subset A of the current space (see Figure 1), or it may be controlled. We shall in this context consider autonomous and controlled switching as applied in the control of a hydraulic cylinder.

2.2. Example: A hydraulic actuator

Figure 2 illustrates a hydraulic actuator. The two main components are the piston and the servo valve regulating the flow of hydraulic oil. The position (x) of the piston is the subject of control while the control input (u) determines the valve position.

Mathematical models of such an actuator are complex and cannot be described accurately by a single linear relationship. Therefore a set of linear submodels are developed, each describing the model of the hydraulic servo-actuator in a subset of the state space (see [1] for details). The submodels are derived through linearization of a complex non-linear model around an operating point in each subset of the state space resulting in 6 submodels or modes:

$$\dot{x}(t) = A_i x(t) + B_i u(t) \quad (3)$$

where we call the solution $x_i(t)$. For each mode, a control algorithm is developed:

$$u_i(t) = K_{p_i} \cdot (r(t) - x(t)) + K_{ff_i} \cdot \dot{r}(t) \quad (4)$$

where $r(t)$, $\dot{r}(t)$ and $x(t)$ are the reference position, the reference velocity and the measured position of the actuator, respectively.

The general structure of the 6 control algorithms consists of a proportional feedback part, parametrized by K_{p_i} (the proportional gain), and a reference velocity feedforward part, parametrized by K_{ff_i} (the feed-forward gain). (K_{p_i}, K_{ff_i}) are the applicable parameters for the control algorithm when the plant is in mode i.

The modes are represented by overlapping subspaces of the plant state space. An autonomous switch of the vector field is detected by evaluating the model error e_i over a period T:

$$e_i = \int_t^{t+T} (x(\tau) - x_i(\tau))^2 d\tau \quad (5)$$

where i ranges over the six modes and x_i denotes the model state. The transition is done when the minimal e_i changes. The autonomous switching set A is thus defined implicitly.

When an autonomous switch is detected (by evaluation of the e_i's) the parameters of the control algorithm should be changed to match the new plant mode. That is, in response to a detected autonomous switch a controlled switch of control algorithm is performed.

2.3. Framework parameters

A software framework for implementing hybrid control systems needs to include facilities for detection of autonomous switching of the plant known as *important events* [10]. The way important events are detected will depend on the application and we want in our framework to be as loose as possible about this to strengthen general applicability. For this reason the detection algorithm is a parameter of the framework. The input to the algorithm is however fixed by the framework to be the stream of sampled values observed from the plant.

On the basis of detected important events the controller may choose to change the control algorithm. This is for example the intended response to an autonomous switch in the hydraulic actuator described above. The selection algorithm varies with the application and is thus provided to the framework as a parameter. A third parameter is of course the set of control algorithms among which this algorithm may choose.

Change of control algorithm need not be caused by an important event but may be *planned* in advance by the fourth parameter of the framework. Given a high level objective of the controller, say a reference curve to follow, it is the job of the planning algorithm to plan switches between control algorithms to exploit their individual characteristics (like speed, stability etc.) in obtaining the aim.

3. Program design

Feedback control of a dynamical plant using a computer is basically a loop ([3]):

```
WHILE TRUE
  SEQ
    sensor ? m
    u:= control.alg(m,r)
    actuator ! u
```

The sensor inputs an estimate `m` of the current state of the plant at sampling time. The estimate is made available to the computer by the use of AD-converters. A control algorithm calculates on basis of the current state of the plant and the reference `r` a new control input to the plant. The control input is applied through an actuator by the use of DA-converters.

Given the hybrid control phenomena discussed in section 2 two major considerations must be made in the elaboration of an architecture for a hybrid control program framework:

- How do we interleave the periodic execution of the feedback control loop with the aperiodic switching between control algorithms, such that the sampling rate is maintained and each switch is performed "immediately" after its conditions become satisfied?
- How do we facilitate a clear and easy-to-use interface to the control engineer such that the program framework is smoothly instantiated with suitable control algorithms, algorithms for detection of and reaction upon important events and so forth?

Our approach is based on the architectural ideas of Nadjm-Tehrani ([11]). She proposes a layered system architecture which provides a clear separation between high level planning and switching mechanisms executing in an aperiodic manner and low level control algorithm and plant interface components executing periodically. This hierarchical architecture allows the combination of two diverse intelligent control philosophies, namely the one in which actions are taken on the basis of brute-force calculations in a centralized world model, and the one in which actions are determined (in, quoting, "rapid and small doses of computation") on the basis of here-and-now information about the system. The former could involve a pre-planning activity and the latter could involve the detection of and reaction upon important events as introduced in section 2.3. Preliminary experiments with this architectural model are described in [8] and [1].

The layered architecture model consists of three layers: Analysis, Rule and Process layer. Each layer consists of one or more functional blocks. Nadjm-Tehrani gives a loose and informal description of the purpose of each block (see [11]), and this description has given us an intuition as to how we design a program architecture that satisfies the aforementioned considerations. The actual development has been a formal one using Duration Calculus ([5, 13]) and the implementation has been written in occam as our experimental facility (a hydraulic robot) uses a network of transputers as hardware platform ([6] shows the formal development and illustrates how the program framework was implemented on that basis). In Figure 3 is shown the resulting architecture as a set of concurrent communicating processes and their channel connections. Furthermore, the placement of the processes with respect to the layers and functional blocks of the architectural model is shown. We will in the following give a short informal description of the different processes.

3.1. Process overview

Starting at the bottom of the architecture in the Process layer, the interface to the plant is implemented by the processes *Estimator* and *Adaptor*. The *Characterizer* process

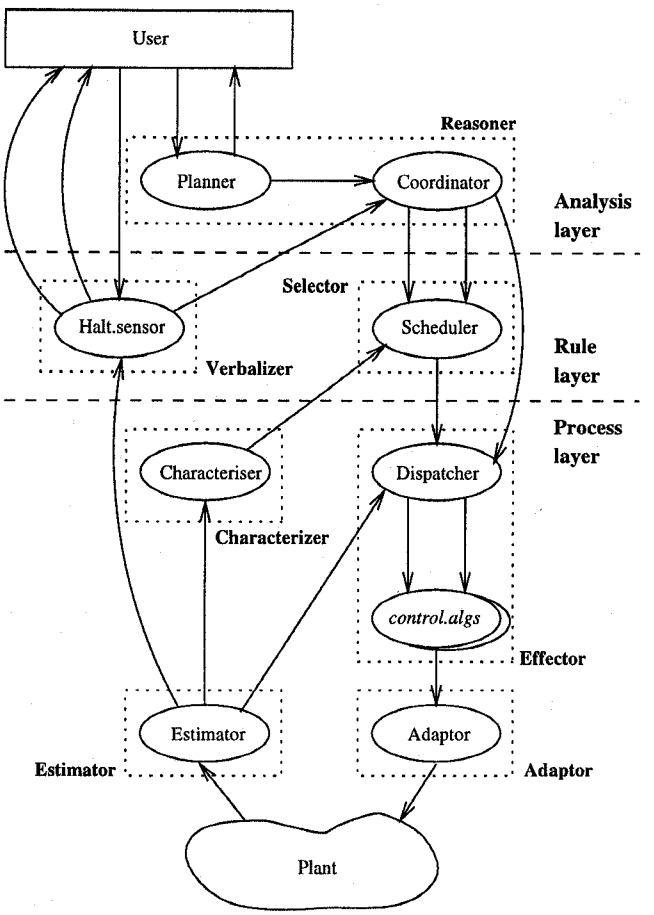

Figure 3. Program architecture

detects important events. The Effector module has the *Dispatcher* process that performs the switches between control algorithms and keeps track of which control algorithm is in control of the plant. Only the control algorithm actually controlling the plant is given measurements and reference-values at sampling time. Control algorithms *control.algs* are implemented as processes and reside in the Effector block as well. We thus find the feedback control loop implemented by *Estimator, Dispatcher*, a control algorithm and *Adaptor*.

In the Rule layer, *Halt.sensor* monitors the conditions under which it is safe to control the plant according to a given reference. Such conditions could for instance depend on the (measured) behaviour of the plant and on a "run" signal from the user. In case the state of the conditions changes, *Coordinator* is notified. *Scheduler* determines when a switch of control algorithm is required. This is the case if a planned switch is timed out or as a reaction to an important event received from *Characterizer*. Performing a switch means that *Scheduler* selects/generates a suitable set of *parameters* for the selected control algorithm and notifies *Dispatcher* of the switch. *Dispatcher* then sends the parameters on to the selected control algorithm (process) and regards the selected control algorithm as the one controlling the plant. This is in fact the core of the interleaving of the feedback control loop with the switching events.

The analysis layer consists of one functional block only, the Reasoner module. In our implementation the *Planner* and *Coordinator* processes constitute Reasoner. *Planner* monitors changes of the reference and performs a pre-planning of switches. *Coordinator* is notified of the result of the planning activity and determines on basis of this and the input from *Halt.sensor* the required control *job*. A job consists of a list of pre-planned switches between control algorithms and a reference. The former is sent to *Scheduler* and the latter is sent to *Dispatcher*.

This section has given an overview of the architecture of our hybrid control program framework. In the beginning of the section we stated the interleaving of aperiodic switching of control algorithm in the otherwise periodic execution of the control loop and the wish for a clear interface to the control engineer as two major considerations that were made in the development of the described architecture and in this subsection we have explained how the first consideration has been taken care of. In the following section we shall go into details and show how the interface to the control engineer looks and is used.

4. Interface

In this section we shall be quite specific about the interface of the program framework to the control engineer. We illustrate the parameters of the interface as pieces of occam code taken from the program framework. It should be noted that the type system of occam is quite primitive. Apart from the primitive types (Booleans, integers and reals) only arrays are available. Composite data-types have to be encoded.

Recall from section 2.3 the four major parameters of the interface:

1. A planning algorithm for á priori planning of switches.
2. An algorithm for the detection of important events.
3. A selection algorithm to react upon important events.
4. A set of control algorithms.

Starting with the last parameter, a control algorithm is implemented as a separate process. This, again, gives the advantage of a possible internal state which is often used in control algorithms, e.g. when numerical integration is used.

```
PROC Control.alg
  (CHAN OF PARAMETERS parameters,
   CHAN OF CA.INPUT ca.input,
   CHAN OF CA.OUTPUT ca.output)
```

```
WHILE TRUE
  ALT
    parameters ? param.l::param  -- start up
      ...
    ca.input ? r; m
      ...
      ca.output ! f(r, m)
      ...
```

The control algorithm processes are each identified as an *index*, and the specific algorithm is parametrized with a set of parameters. The selection of a control algorithm thus consists of choosing a process index and sending a suitable set of parameters on its `param` channel. Consequently, a switch of control algorithm does not necessarily entail a switch of control algorithm process. A control algorithm process receives its parameters as it is selected and will from then on and until another is selected receive measurements and reference values from *Dispatcher* on the channel `ca.input` and produce outputs on the channel `ca.output`.

A switch of control algorithm is either pre-planned or provoked by an important event. The pre-planning activity resides in the *Planner* process, and the concrete interface is the subroutine, `plan`:

```
PROC plan ([Max.band.size]REAL32 xref,
           [2][Max.no.switches]INT sl,
           [Max.band.size]REAL32 B,
           [Infosize]INT info)
```

Input is the reference (`xref`) and output is a list of pre-planned switches (`sl`), a possibly modified reference (`B`) and some auxiliary information (`info`). Given the architecture of the program framework and the signature of the `plan` subroutine, the pre-planning activity is based on static knowledge about the control algorithm characteristics. The process index and the set of parameters of a pre-planned switch are thus selected from a fixed range of possible values. This fixed range is organized in two tables and these tables are a part of the interface as well:

```
VAL [][]INT Index.no.param IS
  [[0,1], [1,2], [2,4]] :

VAL [][]REAL32 Params IS
  [[100.0, Dummy, Dummy, Dummy],
   [ 50.0,  20.0, Dummy, Dummy],
   [ 45.0,  10.0,   5.0,   0.5]] :
```

An element in `sl` now consists of the time for a planned switch and a key to the table, `Index.no.param`. An element of `Index.no.param` consists of an index of a control algorithm process and the number of applicable parameters. These parameters are fetched from `Params` using the same key.

It is not entirely clear how algorithms for detection of important events generally work (is an internal state a necessity, what does an important event "look like" etc.). We have therefore dedicated the *Characterizer* process of the framework to the instantiation of a detection algorithm. This makes it possible to maintain an internal state. The "skeleton" looks as follows:

```
PROC Characterizer(CHAN OF C c.input,
                   CHAN OF I.E important.event)
  WHILE TRUE
    c.input ? r; m
      ...
      IF ...                -- if event detected
        important.event ! ie.r; ie.i
:
```

Characterizer must be ready to accept input in every sampling period in order not to block the control loop (cf. section 3). An important event is output on the channel, `important.event`, and consists in the framework of a variable length stream of floating point numbers and a ditto of integers.

The last major parameter of the interface is the mechanism for reacting upon important events. The reaction is a selection of a (possibly) different control algorithm and the parameter subroutine, `idsyn` (short for *identify and synthesize*), is instantiated with the concrete selection algorithm:

```
PROC idsyn ([Ie.r]REAL32 ie.r,
            [Ie.i]INT ie.i,
            INT ca.i,
            [Max.no.param]REAL32 param)
```

The input to `idsyn` is the information received from *Characterizer* in an important event (`ie.r` and `ie.i`). The output is an index `ca.i` of a control algorithm and a suitable set of parameters `param` for the algorithm. These parameters can either be selected from the tables, `Index.no.param` and `Params`, or as mentioned earlier be generated on basis of the information given in the important event.

4.1. Timing aspects

Given a concrete plant and a concrete hardware platform (assumed to be a network of transputers) the computer engineer is responsible for *configuring* the control program for the hardware. That is, the processes of the architecture (see Figure 3) must be placed on the available transputers. The control program - having real-time constraints stemming, for instance, from the sampling rate - should be configured in a way such that the highest possible percentage of a sampling period is left for calculations to be performed in the algorithms supplied by the control engineer.

As an example, an estimate of the lower bound on the available computation time for a control algorithm, t_{calc} can be found as follows (we assume that only one transputer is available):

$$t_{calc} = T_{samp} - \frac{\#\text{Processes on transputer}}{\#\text{Processes in control loop}} \cdot T_{cycle}$$

where T_{samp} is the sampling period and t_{cycle} is the time spent in the processes of the control loop during one sampling period. t_{calc} is an important parameter of the interface constraining the possible choices of control algorithms. An analysis must be made to ensure that each of the control algorithms supplied has a calculation time below t_{calc}.

A more detailed discussion on this subject falls outside the aim of this paper, but examples of the technique in use are given in [8] and [6].

4.2. Example: A hydraulic actuator

The example given in Section 2.2 serves to illustrate the use of the interface. First of all, we see from (4) that only one control algorithm process is necessary, as it is parametrized with values K_p and K_{ff}. The applicable parameter sets are determined á priori by the control engineer, and `Index.no.param` and `Params` will consequently each have 6 elements (one for each mode).

Detection of the current mode is done using (5). That is, with a period of T a (possibly) new mode is detected and this (possibly) causes a reaction (a switch of control algorithm). We thus identify a detection of a switch of mode as an important event. *Characterizer* is thus instantiated with the discrete version of (5). Once every T time-units the newly detected mode is compared with the current one, and in case they differ an important event has been detected and the new mode number is given to *Scheduler*.

In *Scheduler* `idsyn` is responsible for selecting a suitable control algorithm corresponding to the detected mode. This is simply done by using the mode number received from *Characterizer* as key value for the `Index.no.param` and `Params` tables. Thus the controlled switch is accomplished.

We do not use the pre-planning facility in this example.

5. Conclusion

We have presented the concepts of hybrid systems: plant models, control algorithms, autonomous and controlled switching, cf. [4]. They are illustrated by the concrete example of a hydraulic cylinder. The concepts have been used to define a precise, yet general interface to a program architecture for complex control [11]. The architecture has been implemented on a transputer network in the occam language.

The platform has been successfully tested in an experimental robot setting, with decentralized hybrid (mode switched) control of two individual links. There is thus some evidence that the framework reported here may be successfully used to program complex control systems while maintaining a precise interface to hybrid control theory.

A point where much further work is needed is adaptation of control synthesis techniques to the hybrid framework, e.g. what happens to stablility under mode switching.

References

[1] T. O. Andersen, F. Conrad, A. P. Ravn, T. J. Eriksen, and M. Holdgaard. Mode-switching in hydralic actuator systems - an experiment. In *Proceedings of Eigth Bath International Fluid Power Workshop*. Research Studies Press, 1995. Bath, United Kingdom, September 1995.

[2] N. C. Audsley, A. Burns, R. I. Davis, K. Tindell, and A. J. Wellings. Fixed priority scheduling: an historical perspective. *J. Real-Time Syst.*, 8:173–198, 1995.

[3] J. Billingsley. *Controlling with computers: control theory and practical digital systems*. McGraw-Hill, 1989.

[4] M. S. Branicky, V. S. Borkar, and S. K. Mitter. A unified framework for hybrid control. In *Proc. 33rd Conf. Decision and Control*, pages 4228–4234. IEEE Press, 1994. Lake Buena Vista, FL.

[5] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Proc. Letters*, 40(5), December 1991.

[6] T. J. Eriksen and M. Holdgaard. Development of an intelligent control. Master's thesis, Department of Information Technology, Technical University of Denmark, 1996.

[7] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems*, volume 736 of *LNCS*. Springer-Verlag, 1993.

[8] M. Holdgaard, T. J. Eriksen, and A. P. Ravn. A distributed implementation of a mode switching control program. In *Proceedings of 7th Euromicro Workshop on Real-Time Systems*. IEEE Computer Society Press, 1995.

[9] D. G. Luenberger. *Introduction to Dynamic Systems. Theory, Models & Applications*. Wiley, 1979.

[10] Z. Manna and A. Pnueli. Verifying hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 4–35, 1993.

[11] S. Nadjm-Tehrani. *Reactive Systems in Physical Environments*. PhD thesis, Dept. Comp. and Inf. Science, Linköping University, Sweden, May 1994. Linköping Studies in Science and Technology, Dissertation no. 338.

[12] A. Nerode and W. Kohn, editors. *Hybrid Systems II*, volume 999 of *LNCS*. Springer-Verlag, 1995.

[13] A. P. Ravn. *Design of Embedded Real-Time Computing Systems*. Dept. of Computer Science, DTU, 1995. Doctoral thesis.